Numerical $\mathbb{M}$ATh

Michael Junk
University of Constance
78457 Konstanz, Germany
michael.junk@uni-konstanz.de

Stefan Hölle
University of Constance
78457 Konstanz, Germany
stefan.hoelle@uni-konstanz.de

Abstract

Numerical mathematics is concerned with the representation of general mathematical objects (e.g. the solution of some differential equation) in terms of a limited collection of simpler objects like the basic arithmetic functions on $\mathbb{R}$. Traditionally, the correctness of such representations is rigorously investigated but when it comes to the implementation, error-prone representation changes to data structures are not checked in a formal manner. To avoid this problem, mathematical models of data structures and algorithms are required to enable rigorous arguments down to the level of actual implementation steps. We have carried out this process for various problems in the area of differential equations. In order to keep proofs about the correctness of implementations manageable, intermediate data structures with high reusability are constructed.

1 Background

The project $\mathbb{M}$ATh as acronym for ***m**eta-language for **m**odels, **a**lgorithms and **th**eories* [1, 2, 3] originated in an attempt to combine various aspects of high level math education in a common formal framework. The syntactical elements and rules of the formalism have been designed with two goals: They should be easy to convey even to first year students and they should allow to specify and analyze mathematical models intuitively and consistently in the whole range from word problems in elementary school up to full fledged models in industrial projects. However, to cover all task areas of applied mathematics and scientific computing, also the implementation of numerical algorithms has to be addressed in math education.

The goal of this article is to present our experience with integrating implementation and explanation of numerical algorithms into a framework which supports mathematical modeling and reasoning. While the basic spirit is similar to many existing approaches which combine reasoning and computation like Hoare's approach [4], the Coq-language [5] or combinations of theorem proving and symbolic computing [6, 7] (to name just a few), we concentrate particularly on how algorithms may be formulated to alleviate the reasoning process.

2 Models and programs

In this article, we use the word *model* to denote a *description of a system (i.e. a collection of objects with mutual relations) in a formal language.* For example, a mathematical model of the solar system S describes

In: C. Kaliszyk, E. Brady, J. Davenport, W.M. Farmer, A. Kohlhase, M. Kohlhase, D. Müller, K. Pąk, and C. Sacerdoti Coen (eds.): Joint Proceedings of the FMM and LML Workshops, Doctoral Program and Work in Progress at the Conference on Intelligent Computer Mathematics 2019 co-located with the 12th Conference on Intelligent Computer Mathematics (CICM 2019), Prague, Czech Republic, July 8–12, 2019, published at http://ceur-ws.org

the objects (the sun and planets) in terms of scalar (mass) parameters and vector valued functions of a scalar (time) variable for positions and velocities. The mutual relations are then coded as a set of ordinary differential equations (Newton's law with gravitational forces). From a mathematical point of view, the model (call it M_{phy}) appears as a *theory* which, according to [8], comprises of a finite number of abstract mathematical objects whose interplay emerges from a list of axioms. Once the theory is fixed, consequences may be drawn from the axioms according to precisely stated rules.

In this context, an approximation of the ordinary differential equation by a discrete equation can be considered as a model (call it M_{num}) of M_{phy}. Differences between M_{phy} and M_{num} are due to additional (discretization) parameters, changes in the equations which relate the objects and possibly also in the object representation (the domain of the (time) variable may be restricted to a finite set, for example).

Finally, if a solver for the discrete equations is implemented, the resulting program P is also a model, because it describes M_{num} in a formal language: The mathematical objects are represented by appropriate data structures and the mutual relations are coded in the solution algorithm. The resulting cascade of models is depicted in Figure 1.

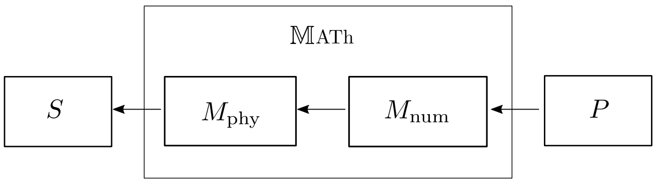

Figure 1: Illustration of the modeling process. Each arrow points from the model to the corresponding system.

Since M_{phy} involves various abstractions and approximations of S, *modeling errors* can occur which means that predictions generated from M_{phy} differ from corresponding observations in S. The modeling error between M_{num} and M_{phy} is traditionally denoted *approximation error* and we would call M_{num} a reasonable model, if the approximation error converges to zero under suitable assumptions. Finally, modelling errors between P and M_{num} may be due to neglected integer overflow or round-off errors (if mathematical integers and reals are modeled by representations of finite bit length), or due to approximations of transcendental by rational functions. Apart from these technical errors also undetected typos and logical mistakes may be part of this modeling error.

3 Reducing errors

In the modeling process described above, the best error control is obtained between M_{num} and M_{phy} when the convergence proof is verified by sufficiently many independent mathematicians or by some trusted proof checker. Besides excluding logical mistakes, the proof also serves as a very careful explanation, *how* M_{num} models M_{phy}.

For the link between M_{phy} and S, such an explanation is generally not possible, because the language describing S is not formal enough to support strict arguments. Nevertheless, the chance of logical errors may be reduced if the concepts used to describe S are closely matched by notions available in the mathematical language so that the translation is simplified. Conversely, the bigger the difference in vocabulary, the more (uncontrolled) reasoning is required to explain the relation between S and M_{phy} which impedes comprehensibility and increases the chance of mistakes (in view of this, one educational objective in our modeling classes is the design of *comprehensible* models: First, mathematical notions are defined which translate key words of the problem – then the problem is reformulated using these terms).

The same idea applied to the link between M_{num} and P calls for some library with data structures and algorithms adapted to the counterparts in M_{num}. The use of such a library may turn the program into a comprehensible model of M_{num} and thus reduce the chance of mistakes. To increase confidence in the correctness of P, time consuming tests may be carried out. However, they are often incomplete and therefore much less satisfactory [9] than a proof of how P relates exactly to M_{num}. To formulate such a proof, one of the two formal languages has to be strong enough to speak about the other. Sometimes, stronger languages are constructed by enriching the programming languages (e.g. C or Java) with a specification syntax (e.g. ACSL or JML). Then code analysis is possible with reasoning systems [10, 11].

In numerical math courses, algorithms are generally presented in some pseudo-code which combines mathematical notation with a small selection of control structures common to most programming languages. Data types like `int`, `real` or `list` are identified with $\mathbb{Z}, \mathbb{R}$ and Cartesian powers respectively. In this way, restriction to a single language is avoided and notation stays closer to common mathematical usage.

We pick up the latter approach by first describing certain aspects of (a family of) computer languages L in terms of a mathematical model M_{cmp} (here the objects are data structures and algorithms). Then (see figure 2), we reformulate M_{num} using these abstract objects and prove mathematically, how the resulting model M_{alg} corresponds to M_{num}. Finally, if M_{alg}-expressions, which are based solely on M_{cmp} objects, are automatically translated into some L-coherent programming language, even translation errors can be avoided.

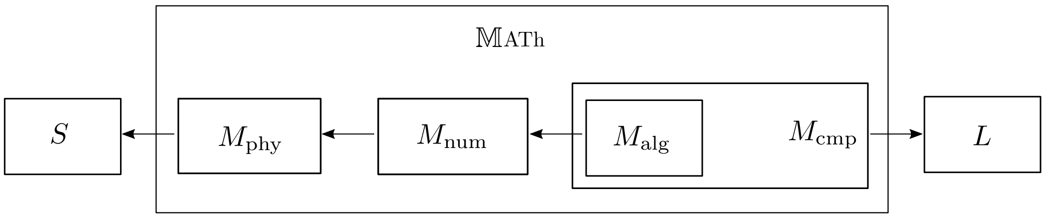

Figure 2: A modeling approach with most steps contained in the same mathematical framework.

Since the modeling process is very flexible, we can include high level algorithms into M_{cmp} which simplifies the formulation of M_{alg} but reduces mathematical control. Conversely, rigor and effort in the formulation of M_{alg} is increased if the model is based on low level algorithms. Similarly, the model assumptions on the data structures can range from IEEE 754 rules to a mere identification of computer and mathematical reals if round-off errors are considered less critical.

4 A glimpse into M_{cmp}

We have worked out an exemplary version of M_{cmp} which starts with the fundamental object *rawData* representing a set whose elements are interpreted as data objects in L.

In order to model the binding of raw data to mathematical concepts, we could postulate a mapping from rawData to objects defined in 𝕄ATh-theories. However, such a global binding map would cause some difficulty: Since the development of numerical algorithms generally comes along with the definition of new data structures for relevant mathematical objects, new axioms on the binding map are required during the development. In contrast to usual mathematical practice, it would be impossible to collect *all* axioms of M_{cmp} before drawing conclusions.

To avoid this, we favor a less global approach and introduce *representations* which bind rawData elements to mathematical meaning in records with a *data* field for the raw data and an *obj* field for its mathematical interpretation. Formally, the 𝕄ATh-definition of this notion would be part of M_{cmp}.

representation := `r` : record **with** dom(`r`) ⊂ {'data', 'obj'}; `r`.data : rawData □

Collections of representations whose mathematical interpretations range in some set X with no rawData object connected to more than one interpretation will be called data types of X. More specifically,

dataType(X) := T **with** T ⊂ representation; ($x \in T \mapsto x$.data) : injective; ∀`x` ∈ T **holds** `x`.obj ∈ X □

Using this notion, the abstract M_{cmp}-parameter `real` is introduced with the axiom `real` : dataType($\mathbb{R}$). This assumption does not entail that each real number is represented by some raw data which would, however, follow from the axiom objRange(`real`) = $\mathbb{R}$. Here objRange(T) is defined as the set of all objects corresponding to the elements of the dataType T.

To describe functions from rawData to rawData which are realizable in L, the notion *rawMapping* is introduced as a recursively defined set: It contains identity and constant maps as explicit elements and certain rules allow the creation of new raw mappings from given ones (e.g. by composition). An *algorithm* is then defined as a representation-valued mapping on some dataType with the property that the induced map between the data fields of arguments and images is a rawMapping. In this way, the recursive construction principle is passed on from raw mappings to algorithms.

While algorithms give rise to raw mappings when representations are restricted to their data fields, they may also be considered as functions between mathematical interpretations. This requires, however, that representations of the same object are mapped to a common mathematical interpretation. In this case, we speak of algorithmic representations

algorithmicRepresentation(f) := `f` : algorithm **with** ∀`x` ∈ dom(`f`) **holds** `f`(`x`).obj = f(`x`.obj) □

Altogether, there are three levels on which an algorithmic representation can operate. On the level of *rawData*-objects as a *rawMapping*, on the level of *representation*-objects as an *algorithm* and on the level of purely mathematical objects as a mathematical function.

In our model, algorithmic representations are used to relate abstract algorithms like `plus` on `real` to their mathematical counterparts (e.g. with the axiom `plus` : algorithmicRepresentation(plus) the relation to the addition function plus on $\mathbb{R}$ is established). Moreover, in M_{alg}, certain algorithms are proved to be algorithmic representations of functions in M_{num} which serves as careful explanation of the relation between M_{alg} and M_{num}.

When modeling compound data types of L like lists or records, one could either follow the approach above or use related types in 𝕄ATh directly. For example, a simple way to axiomatize record formation in L is to consider a 𝕄ATh-record r as rawData element if and only if the images of r (i.e. the indexed values) are rawData.

While such a close relation between L- and 𝕄ATh-records alleviates the use in 𝕄ATh expressions, it may complicate automatic translations to L-compatible languages. This effect is more obvious in the case of lists because the integer indices are distinguished objects in L and 𝕄ATh and 1 as the minimal list index in 𝕄ATh may differ from the choice in other languages. For this reason, we have modeled L-lists without resorting to the 𝕄ATh counterparts directly. Instead, `list` has been introduced as an abstract set of representations in M_{cmp} where each element `x` of `list` represents a 𝕄ATh-list (which is a function on a set $\{1,\ldots,n\}$). In order to enable list usage, additional L objects are required like `length` which maps each `x` $\in$ `list` to a non-negative `int` and `eval` which connects `x` with an algorithmic representation of `x`.obj. Finally, the creation of a `list` from a finite number of representations requires the abstract object `explicitList`. The precise interplay of these objects and other fundamental `list` operations like joining, composing, repeating, transposing or flattening are described with appropriate axioms.

5 A glimpse into M_{alg}

Our experience with algorithms for the approximate solution of ordinary and partial differential equations in bachelor and master courses is that linear algebra aspects like setting up coordinate vectors, functionals and matrices turn out to be primary sources for logical errors. This is particularly the case when ad-hoc representations of finite difference grids, finite element meshes, finite dimensional vector spaces as well as linear, non-linear and discrete functions are formulated in terms of simple `list` structures. In comparison, round-off errors are generally less critical so that details of floating point arithmetic have not been introduced in our model. Instead, we identified `int` and `real` with $\mathbb{Z}$ and $\mathbb{R}$ (see [12, 13] for approaches which focus on correct models of floating point operations).

To give an example, we consider an equidistant rectangular grid with distance $h > 0$ between nodes in both space directions as it may appear in a finite difference discretization. In this case, natural indices are integer pairs $(i,j) \in \mathbb{Z}^2$ to refer to the grid nodes $h \cdot (i,j) \in G$ and the vector space of grid functions $G \to \mathbb{R}$ (also denoted $\mathbb{R}^G$) plays a central role. A useful basis of its dual space consists of the evaluation functionals at the grid nodes so that the natural indexing is also in terms of pairs (i,j). Finite difference operators are simple linear combinations of these basis functionals which typically involve indices of geometrically close nodes.

Since basic arrays in programming languages are indexed by $\{1,\ldots,n\}$ or $\{0,\ldots,n-1\}$ respectively, a bijective index-mapping (resulting, for example, from row- or column-wise node enumeration) has to be employed which renders the relevant expressions more technical and error-prone. This situation is aggrevated if systems of equations with time dependence are considered. In fluid dynamics, for instance, the unknowns may consist of a velocity vector and a pressure value at each node and at each time point so that the relevant function space is $((\mathbb{R}^2 \times \mathbb{R})^G)^T$ (here T is the temporal grid). To realize elements of such structured spaces, we do not translate them directly to lists but introduce reusable intermediate data structures and algorithms: representations of finite sets with operations like product, intersection or union, representations of functions between finite sets with evaluation and composition, representations of functions from finite sets into general data types and representations of vector space operations on arbitrary finite products of `real`.

Establishing all these structures and proving their relationship to the mathematical counterparts amounts to the construction of a generalized software library with a very precise documentation and high reusability. The actual formulation of the target algorithms uses this library and the associated mathematical theory alleviates the proof of the theorems which connect the algorithms to the original mathematical problems. Apart from teaching code generation which is particularly amenable for program verification, we are also planning a tool which creates code automatically from logically verified 𝕄ATh-algorithms in different programming languages. So far, we have tested this translation process manually with Scala implementations of the algorithms.

References

[1] 𝕄ATh-Homepage, `http://www.math.uni-konstanz.de/mmath/en`.

[2] Junk, M., Hölle, S., Sahli, S.: *Formalized Mathematical Content in Lecture Notes on Modelling and Analysis.* In: Rabe, F., Farmer, W. M., Passmore, G. O. and Youssef, A. (eds.) Intelligent Computer Mathematics. Lecture Notes in Computer Science, vol. 11006, pp. 125-130. Springer (2018)

[3] Junk, M., Hölle, S., Sahli, S.: *A Meta Language for Mathematical Reasoning.* In: Osman Hasan, O., Kaliszyk, C., Naumowicz, A. (eds.) Proceedings of the Workshop Formal Mathematics for Mathematicians (FMM), Hagenberg, Austria (2018)

[4] Hoare, C.A.R.: *An Axiomatic Basis for Computer Programming.* In: Communications of the ACM 12, 576–580 (1969)

[5] Bertot, Y., Castéran, P.: *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science, Springer, 2004.

[6] Homann, K., Calmet, J.: *Combining Theorem Proving and Symbolic Mathematical Computing.* In: J. Calmet and J. A. Campbell (eds.): Integrating Symbolic Mathematical Computation and Artificial Intelligence, Vol. 958 of Lecture Notes in Computer Science (1995).

[7] Carette, J., Farmer, W. M., Sharoda, Y.: *Biform Theories: Project description.* In: F. Rabe, W. M. Farmer, G. O. Passmore, and A. Youssef, eds., Intelligent Computer Mathematics (CICM 2018), Lecture Notes in Computer Science (LNCS), 11006:76–86, 2018.

[8] A. Tarski. *Introduction to Logic and to the Methodology of Deductive Sciences.* Oxford University Press, 1941.

[9] Dijkstra, E. W. *The humble programmer*, Communications of the ACM, vol 15, 859–866 (1972)

[10] Kirchner, F., Kosmatov, N., Prevosto, V.; Signoles, J., Yakobowski, B. *Frama-C, A Software Analysis Perspective.* In Formal Aspects of Computing, vol. 27, Springer (2015).

[11] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P. H., Ulbrich, M. *Deductive Software Verification - The KeY Book - From Theory to Practice.* Lecture Notes in Computer Science 10001, Springer (2016).

[12] Sylvie Boldo, S., Clément, F., Filliâtre, J.-C., Mayero, M., Melquiond, G., Weis, P. *Trusting computations: a mechanized proof from partial differential equations to actual program.* Computers and Mathematics with Applications, vol. 68, Elsevier (2014).

[13] Immler, F. *A Verified ODE Solver and Smale's 14th Problem.* Dissertation, Technische Universität München (2018).